Chapter 5

Advanced C# Class Construction Techniques

This chapter rounds out your introduction to the core aspects of the C# language by examining a number of advanced (but extremely useful) syntactic constructs. To begin, you learn how to construct and use an *indexer method*. This C# mechanism enables you to build custom types, which exposes internal subtypes using the familiar bracket operator (i.e., []). If you have a C++ background, you will find that creating a C# indexer method is analogous to overloading the [] operator on a C++ class. Once you learn how to build an indexer, you then examine how to overload various operators (+, -, <, > and so forth) for a custom C# type.

This chapter then examines three techniques that enable the objects in your system to engage in bidirectional communications. First, you learn about the C# "delegate" keyword, which is little more than a type-safe function pointer. Once you learn how to create and manipulate delegates, you are in a perfect position to investigate the .NET event protocol, which is based on the delegation model. Finally, you discover how the use of custom interfaces can also enable bidirectional communications (which should ring a bell for those coming from a COM background).

I wrap up by examining how you can document your types using XML attributes, and how the Visual Studio.NET IDE automatically generates Web-based documentation for your projects. Although this might not qualify as a truly "advanced" technique, it is a high note on which to end the chapter.

Building a Custom Indexer

At this point, you should feel confident building C# types using traditional OOP (refer to Chapter 3) as well as interface-based programming techniques (refer to Chapter 4). In this chapter, I take some time to examine some additional aspects of C# that you may not be readily failure with, beginning with the concept of an *indexer*.

Most programmers (such as yourself) are very familiar with the process of accessing discrete items held within a standard array using the index (aka *bracket*) operator:

```
// Declare an array of integers.
int[] myInts = {10, 9, 100, 432, 9874};

// Use the [] operator to access each element.
for(int j = 0; j < myInts.Length; j++)
     Console.WriteLine("Index {0} = {1}", j,  myInts[j]);
```

The C# language supports the capability to build custom classes that may be indexed just like an array of intrinsic types. It should be no big surprise that the method that provides the capability to access items in this manner is termed an "indexer."

Before exploring how to create such a construct, let's begin by seeing one in action. Assume you have added support for an indexer method to the Cars container developed in the previous chapter. Observe the following usage:

```
// Indexers allow you to access items in an array-like fashion.
public class CarApp
{
    public static void Main()
    {
        // Assume the Cars type has an indexer method.
        Cars carLot = new Cars();

        // Make some cars and add them to the car lot.
        carLot[0] = new Car("FeeFee", 200, 0);
        carLot[1] = new Car("Clunker", 90, 0);
        carLot[2] = new Car("Zippy", 30, 0);

        // Now obtain and display each item.
        for(int i = 0; i < 3; i++)
        {
            Console.WriteLine("Car number {0}:", i);
            Console.WriteLine("Name: {0}", carLot[i].PetName);
            Console.WriteLine("Max speed: {0}", carLot[i].MaxSpeed);
        }
    }
}
```

A test run would look something like Figure 5-1.

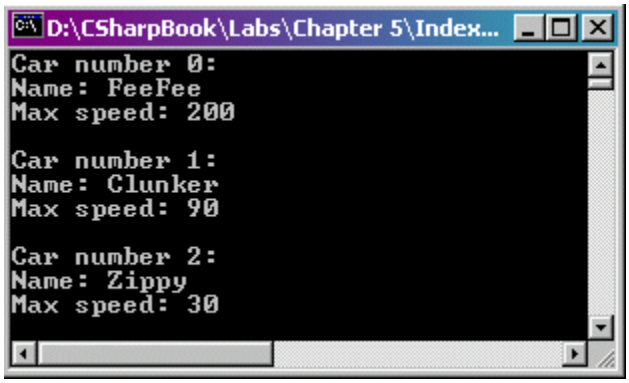

Figure 5-1. Accessing cars using an indexer

As you can see, indexers behave much like a custom collection supporting the IEnumerator and IEnumerable interfaces. The only major difference is that rather than

accessing the contents using interface references, you are able to manipulate the internal collection of automobiles just like a standard array.

Now for the big question: How do you configure the Cars class (or any class) to do so? The indexer itself is represented as a slightly mangled C# property. In its simplest form, an indexer is created using the this[] syntax:

```
// Add the indexer to the existing class definition.
public class Cars : IEnumerator, IEnumerable
{
…
    // Let's rollback to the basics and simply make use of a standard array
    // to contain the cars.  You are free to use an ArrayList if you desire…
    private Car[] carArray;

    public Cars()
    {
        carArray = new Car[10];
    }

    // The indexer returns a Car based on a numerical index.
    public Car this[int pos]
    {
        // Accessor returns an item in the array.
        get
        {
            if(pos < 0 || pos > 10)
                throw new IndexOutOfRangeException("Out of range!");
            else
                return (carArray[pos]);
        }
        // Mutator populates the array.
        set { carArray[pos] = value;}
    }
}
```

Beyond the use of the “this” keyword, the indexer looks just like any other C# property declaration. Do be aware that indexers do not provide any array-like functionality beyond the use of the subscript operator. In other words, the object user cannot write code such as:

```
// Use System.Array.Length? Nope!
Console.WriteLine("Cars in stock: {0}", carLot.Length);
```

To support this functionality, you would need to add your own Length property to the Cars type, and delegate accordingly:

```
public class Cars
{
    …
    // Containment / delegation in action once again.
    public int Length() { /* figure out number of non-null entries in array. */}
}
```

However, if you are in need of this functionality, you will find your task will be much easier if you make direct use of one of the System.Collections types to hold your internal items, rather than a simple array.

SOURCE CODE The Indexer project is located under the Chapter 5 subdirectory.

Overloading Operators

C#, like any programming language, has a canned set of tokens that are used to perform basic operations on intrinsic types. For example, everyone knows that the + operator can be applied to two integers in order to yield a new integer:

```
// The + operator in action.
int a = 100;
int b = 240;
int c = a + b; // c = = 340
```

This is no major news flash, but have you ever stopped and noticed how the same + operator can be applied to any intrinsic C# data type? For example:

```
// + operator with strings.
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2;   // s3 = = Hello world!
```

In essence, the + operator has been overloaded to function correctly on various individual data types. When the + operator is applied to numerical types, the result is the summation of the operands. However, when applied to string types, the result is string concatenation. The C# language (like C++ and unlike Java) provides the capability for you to build custom classes and structures that also respond uniquely to the same set of basic tokens (such as the + operator). Thus, if you equip a type to do so, it is possible to apply various operators to a custom class.

To keep your wits about you, assume the following simple Point class:

```
// You can't get much lamer than this!
public class Point
{
    private int x, y;
    public Point(){}
    public Point(int xPos, int yPos)
    {
        x = xPos;
```

```
            y = yPos;
        }
        public override string ToString()
        {
            return "X pos: " + this.x + " Y pos: " + this.y;
        }
    }
    s
```

Now, logically speaking it makes sense to add Points together. On a related note, it would be helpful to subtract one Point from another. For example, if you created two Point objects with some initial startup values, you would *like* to do something like this:

```
// Adding and subtracting two Points.
public static int Main(string[] args)
{
    // Make two points
    Point ptOne = new Point(100, 100);
    Point ptTwo = new Point(40, 40);

    // Add the points to make a new point.
    Point bigPoint = ptOne + ptTwo;
    Console.WriteLine("Here is the big point: {0}", bigPoint.ToString());

    // Subtract the points to make a new point.
    Point minorPoint = bigPoint - ptOne;
    Console.WriteLine("Just a minor point: {0}", minorPoint.ToString());

    return 0;
}
```

Clearly, your goal is to somehow make your Point class react uniquely to the + and – operators. To allow a custom type to respond to these intrinsic tokens, C# provides the "operator" keyword, which can only be used in conjunction with *static* methods. To illustrate:

```
// A more intelligent Point class.
public class Point
{
    private int x, y;
    public Point(){}
    public Point(int xPos, int yPos){ x = xPos; y = yPos; }

    // The Point class can be added…
    public static Point operator + (Point p1, Point p2)
    {
        Point newPoint = new Point(p1.x + p2.x, p1.y + p2.y);
        return newPoint;
    }
```

```
        // …and subtracted.
        public static Point operator - (Point p1, Point p2)
        {
            // Figure new X (assume [0,0] base).
            int newX = p1.x - p2.x;
            if(newX < 0)
                throw new ArgumentOutOfRangeException();

            // Figure new Y (also assume [0,0] base).
            int newY = p1.y - p2.y;
            if(newY < 0)
                throw new ArgumentOutOfRangeException();

            return new Point(newX, newY);
        }

        public override string ToString()
        {
            return "X pos: " + this.x + " Y pos: " + this.y;
        }
    }
```

Notice that the class now contains two strange looking methods called *operator* + and *operator* –. The logic behind operator + is simply to return a brand new Point based on the summation of the incoming Point objects. Thus, when you write pt1 + pt2, under the hood you can envision the following hidden call to the static operator + method:

```
// p3 = Point.operator + (p1, p2)
p3 = p1 + p2;
```

Likewise, p1 – p2 maps to:

```
// p3 = Point.operator - (p1, p2)
p3 = p1 - p2;
```

If you were to take your class out for a test run, you would see something like Figure 5-2.

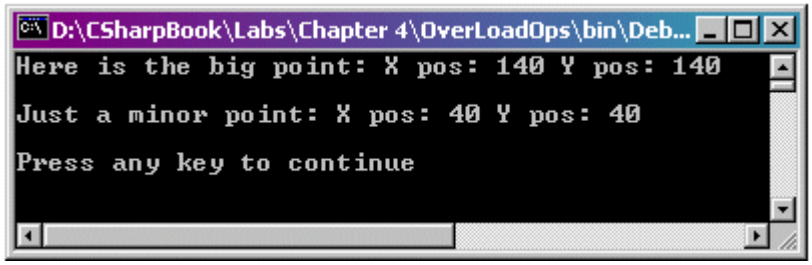

Figure 5-2. Overloaded operators at work

The capability to overload operators is useful in that it enables the object user to work with your types (more or less) like any intrinsic data item. Other languages (such as Java)

do not support this capability. Also understand that the capability to overload operators is *not a requirement* of the Common Language Specification; thus, not all .NET-aware languages support types containing overloaded operators. However, you can achieve the same functionality using public methods. For example, you could write the Point class as so:

```
// Making use of methods rather than overloaded ops.
public class Point
{
…
    // Operator + as AddPoints()
    public static Point AddPoints (Point p1, Point p2)
    {
        return new Point(p1.x + p2.x, p1.y + p2.y);
    }

    // Operator - as SubtractPoints()
    public static Point SubtractPoints (Point p1, Point p2)
    {
        // Figure new X.
        int newX = p1.x - p2.x;
        if(newX < 0)
            throw new ArgumentOutOfRangeException();

        // Figure new Y.
        int newY = p1.y - p2.y;
        if(newY < 0)
            throw new ArgumentOutOfRangeException();

        return new Point(newX, newY);
    }
}
```

You could then add Points as so:

```
// As member f(x)'s
Point finalPt = Point.AddPoints(ptOne, ptTwo);
Console.WriteLine("My final point: {0}", finalPt.ToString());
```

Seen in this light, overloaded operators are always an optional construct you may choose to support for a given class. Remember however, that they are little more than a friendly variation on a traditional public method, and are not CLS-compliant. When you are building production level classes that support overloaded operators, you should always support member function equivalents. To maximize your coding efforts, simply have the overloaded operator call the member function alternative (or vice versa). For example:

```
public class Point
{
…
    // For overload operator aware languages.
    public static Point operator + (Point p1, Point p2)
    {
        return AddPoints(p1, p2);
    }

    // For overloaded challenged languages.
    public static Point AddPoints (Point p1, Point p2)
    {
        return new Point(p1.x + p2.x, p1.y + p2.y);
    }
}
```

Overloading the Equality Operators

As you recall, System.Object.Equals() can be overridden in order to perform value-based (rather than referenced-based) comparisons between objects. In addition to overriding Equals() and GetHashCode(), an object may choose to override the equality operators (= = and !=).To illustrate, here is the updated Point class:

```
// This incarnation of Point also overloads the = = and != operators.
public class Point
{
    public int x, y;
    public Point(){}
    public Point(int xPos, int yPos){x = xPos; y = yPos;}
…
    public override bool Equals(object o)
    {
        if( ((Point)o).x = = this.x &&
            ((Point)o).y = = this.y)
                return true;
        else
                return false;
    }

    public override int GetHashCode()
    { return this.ToString().GetHashCode(); }

    // Now let's overload the = = and != operators.
    public static bool operator = =(Point p1, Point p2)
    {
        return p1.Equals(p2);
    }
```

```
    public static bool operator !=(Point p1, Point p2)
    {
        return !p1.Equals(p2);
    }
}
```

Notice how the implementation of operator = = and operator != simply makes a call to the overridden Equals() method to get the bulk of the work done. Given this, you can now exercise your Point class as so:

```
// Make use of the overloaded equality operators.
public static int Main(string[] args)
{
…
    if(ptOne = = ptTwo)          // Are they the same?
        Console.WriteLine("Same values!");
    else
        Console.WriteLine("Nope, different values.");

    if(ptOne != ptTwo)          // Are they different?
        Console.WriteLine("These are not equal.");
    else
        Console.WriteLine("Same values!");
}
```

As you can see, it is quite intuitive to compare two objects using the well-known = = and != operators rather than making a call to Object.Equals(). As a rule of thumb, classes that override Object.Equals() should always overload the = = and !+ operators.

If you do overload the equality operators for a given class, keep in mind that C# demands that if you override operator = =, you *must* also override operator !=, just as when you override Equals() you will need to override GetHashCode(). This ensures that an object behaves in a uniform manner during comparisons and functions correctly if placed into a hash table (if you forget, the compiler will let you know).

SOURCE CODE The OverLoadOps project is located under the Chapter 5 subdirectory.

Overriding the Comparison Operators

In the previous chapter, you learned how to implement the IComparable interface, in order to compare the relative relationship between two like objects. Additionally, you may also overload the comparison operators (<, >, <= and >=) for the same class. Like the equality operators, C# demands that < and > are overloaded as a set. The same holds true for the <= and >= operators. If the Car type you developed in Chapter 4 overloaded these comparison operators, the object user could now compare types as so:

```
// Exercise the overloaded < operator for the Car class.
public class CarApp
{
     public static int Main(string[] args)
     {
          // Make an array of Car types.
          Car[] myAutos = new Car[5];

          myAutos[0] = new Car(123, "Rusty");
          myAutos[1] = new Car(6, "Mary");
          myAutos[2] = new Car(6, "Viper");
          myAutos[3] = new Car(13, "NoName");
          myAutos[4] = new Car(6, "Chucky");

          // Is Rusty less than Chucky?
          if(myAutos[0] < myAutos[4])
               Console.WriteLine("Rusty is less than Chucky!");
          else
               Console.WriteLine("Chucky is less than Rusty!");
          return 0;
     }
}
```

Because the Car type already implements IComparable (see Chapter 4), overloading the comparison operators is trivial. Here is the updated class definition:

```
// This class is also comparable using the comparison operators.
public class Car : IComparable
{
…
     public int CompareTo(object o)
     {
          Car temp = (Car)o;
          if(this.CarID > temp.CarID)
               return 1;
          if(this.CarID < temp.CarID)
               return -1;
          else
               return 0;
     }
     public static bool operator < (Car c1, Car c2)
     {
          IComparable itfComp = (IComparable)c1;
          return (itfComp.CompareTo(c2) < 0);
     }
     public static bool operator > (Car c1, Car c2)
     {
```

```
            IComparable itfComp = (IComparable)c1;
            return (itfComp.CompareTo(c2) > 0);
        }
        public static bool operator <= (Car c1, Car c2)
        {
            IComparable itfComp = (IComparable)c1;
            return (itfComp.CompareTo(c2) <= 0);
        }
        public static bool operator >= (Car c1, Car c2)
        {
            IComparable itfComp = (IComparable)c1;
            return (itfComp.CompareTo(c2) >= 0);
        }

    }
```

SOURCE CODE The ObjCompWithOps project is located under the Chapter 5 subdirectory.

Final Thoughts Regarding Operator Overloading

As you have just seen, C# provides the capability to build types that can respond uniquely to various intrinsic, well-known operators. Now, before you go and retrofit all your classes to support such behavior, you must be sure that the operator(s) you are about to overload make some sort of logical sense in the world at large.

For example, let's say you overloaded the multiplication operator for the Engine class. What exactly would it mean to multiply two Engine objects? Not much. Overloading operators is generally only useful when building utility types. Strings, points, rectangles, fractions, and hexagons make good candidates for operator overloading. People, managers, cars, headphones, and baseball hats do not. Use this feature wisely.

Also, always remember that not all languages targeting the .NET platform will support overloaded operators for custom types! Therefore, always test your types against any language that may make use of a class defining overloaded operators. If you want to be completely sure that your types will work in any .NET-aware language, supply the same functionality using custom methods in addition to your operator set (as illustrated earlier in this chapter).

Finally, be aware that you cannot overload each and every intrinsic C# operator. Table 5-1 outlines the "overloadability" of each item:

Table 5-1. Valid Overloadable Operators

C# Operator	**Meaning in Life (Can this operator be overloaded?)**
+ - ! ~ ++ -- true false	This set of unary operators can be overloaded.
+ - * / % & | ^ < >>	These binary operators can be overloaded.
= = != > <= >=	The comparison operators can be overloaded. Recall, however, the C# will demand that "like" operators (i.e., < and >, <= and >=, = =, and !=) are overloaded together.
[]	The [] operator cannot technically be overloaded. As you have seen earlier in this chapter, however, the indexer construct provides the same functionality.

Understanding (and Using) Delegates

Up until this point, every sample application you have developed added various bits of code to Main(), which (in some way or another) sent messages *to* a given object. However, you have not yet examined how these objects can *talk back* to the object that created them in the first place. In the "real world" it is quite common for the objects in a system to engage in a two-way conversation. Thus, let's examine a number of ways in which objects can be programmed to do this very thing.

As you may know, the Windows API makes frequent use of function pointers to create entities termed "callback functions" or simply "callbacks." Using callbacks, programmers are able to configure one function to report back to (call back) another function in the application. The problem with standard C(++)callback functions is that they represent nothing more than a simple memory address. Ideally, C(++) callbacks could be configured to include additional type-safe information such as the number of (and types of) parameters, return value, and calling convention. Sadly, this is not the case in traditional C(++)/Win32 callback functions.

In C#, the callback technique is accomplished in a much safer and more object-oriented manner using the "delegate" keyword. When you wish to create a delegate in C#, you not only specify the name of the method, but the set of parameters (if any) and return type as well. Under the hood, the "delegate" keyword represents an instance of a class deriving from System.MulticastDelegate. Thus, when you write:

```
public delegate void PlayAcidHouse(object PaulOakenfold, int volume);
```

the C# compiler produces a new class, which looks something like the following:

```
public class PlayAcidHouse : System.MulticastDelegate
{
    PlayAcidHouse(object target, int ptr);

    // The synchronous Invoke() method.
    public void virtual Invoke(object PaulOakenfold, int volume);

    // You also receive an asynchronous version of the same callback.
    public virtual IAsyncResult BeginInvoke(object PaulOakenfold, int volume,
                                            AsyncCallback cb, object o);
    public virtual void EndInvoke(IAsyncResult result);
}
```

Notice that the class that is created on your behalf contains two public methods that enable you to synchronously or asynchronously work with the delegate (Invoke() and BeginInvoke() respectively). To keep things simple, I will focus only on the synchronous behavior of the MulticastDelegate type.

Building an Example Delegate

To illustrate the use of delegates, let's begin by updating the Car class to include two new Boolean member variables. The first is used to determine if your automobile is due for a wash (isDirty); the other represents if the car in question is in need of a tire rotation (shouldRotate). To enable the object user to interact with this new state data, Car also defines some additional properties and an updated constructor. Here is the story so far:

```
// Another updated Car class.
public class Car
{
…
    // NEW!  Are we in need of a wash? Need to rotate tires?
    private bool isDirty;
    private bool shouldRotate;

    // Extra params to set bools.
    public Car(string name, int max, int curr, bool dirty, bool rotate)
    {
        …
        isDirty = dirty;
```

```
            shouldRotate = rotate;
        }
        public bool Dirty      // Get and set isDirty.
        {
            get{ return isDirty; }
            set{ isDirty = value; }
        }
        public bool Rotate     // Get and set shouldRotate.
        {
            get{ return shouldRotate; }
            set{ shouldRotate = value; }
        }
    }
```

Now, assume you have declared the following delegate (which again, is nothing more than an object-oriented wrapper around a function pointer) within your current namespace:

```
// This delegate is actually a class encapsulating a function pointer
// to 'some method' taking a Car as a parameter and returning void.
public delegate void CarDelegate(Car c);
```

Here, you have created a delegate named CarDelegate. The CarDelegate type represents "some" function taking a Car as a parameter and returning void. If you were to examine the internal representation of this type using ILDasm.exe, you would see something like Figure 5-3 (notice the "extends" informational node).

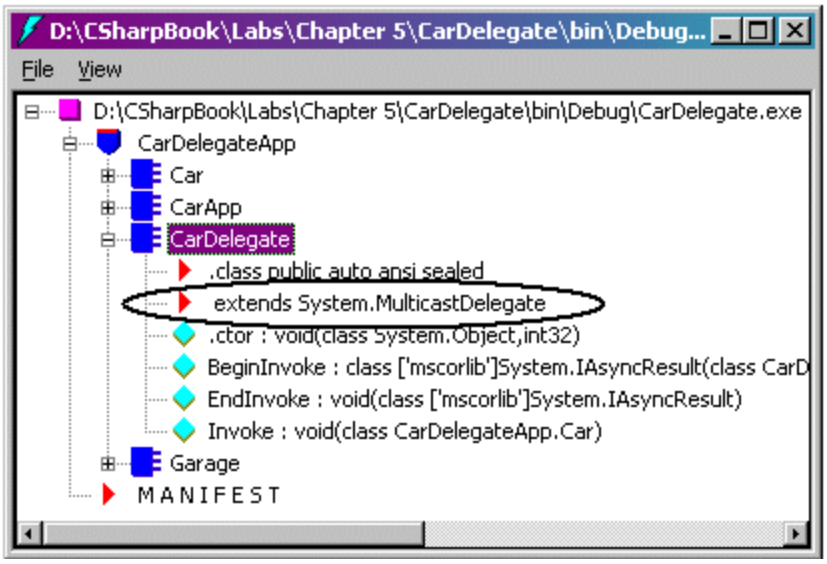

Figure 5-3. C# delegates represent a class deriving from MulticastDelegate.

Delegates as Nested Types

Currently, your delegate is decoupled from its logically related Car type (given that you have simply declared the CarDelegate type within the defining namespace). While there is nothing horribly wrong with the approach, a more enlightened alternative would be to define the CarDelegate directly within the Car class:

```
// This time, define the delegate as part of the class definition.
public class Car : Object
{
    // This is represented as Car$CarDelegate (i.e., a nested type).
    public delegate void CarDelegate(Car c);
…
}
```

Given that the "delegate" keyword produces a new class deriving from System.MulticastDelegate, the CarDelegate is in fact a nested type definition! If you check ILDasm.exe (see Figure 5-4), you will see the truth of the matter.

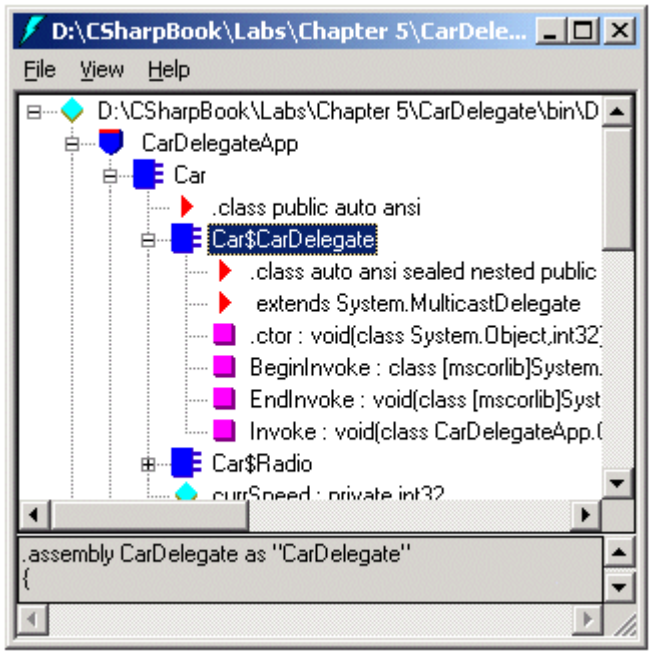

Figure 5-4. Nesting the delegate

Members of System.MutlicastDelegate

So to review thus far, when you create delegates, you indirectly build a type that derives from System.MulticastDelegate (which by the way, derives from the System.Delegate base class). Table 5-2 illustrates some interesting inherited members to be aware of.

Table 5-2. Select Inherited Members

Inherited Member	Meaning in Life
Method	This property returns the name of the method pointed to.
Target	If the method pointed to is a member of a class, this member returns the name of the class. If the value returned from Target equals null, the method pointed to is static.
Combine()	This static method is used to build a delegate that points to a number of different functions.
GetInvocationList()	Returns an array of Delegate types, each representing an entry in the list of function pointers.
Remove()	This static method removes a delegate from the list of function pointers.

Multicast delegates are capable of pointing to any number of functions, because this class has the capability to hold multiple function pointers using an internal linked list. The function pointers themselves can be added to the linked list using the Combine() method or the overloaded + operator. To remove a function from the internal list, call Remove().

Using the CarDelegate

Now that you have a pointer to "some" function, you can create other functions that take this delegate as a parameter. To illustrate, assume you have a new class named Garage. This type maintains a collection of Car types contained in an ArrayList. Upon creation, the ArrayList is filled with some initial Car types.

More importantly, the Garage class defines a public ProcessCars() method, which takes a single argument of type Car.CarDelegate. In the implementation of ProcessCars(), you pass each Car in your collection as a parameter to the "function pointed to" by the delegate.

To help understand the inner workings of the delegation model, let's also make use of two members defined by the System.MulticastDelegate class (Target and Method) to determine exactly which function the delegate is currently pointing to. Here, then, is the complete definition of the Garage class:

```
// The Garage class has a method that makes use of the CarDelegate.
public class Garage
{
    // A list of all cars in the garage.
    ArrayList theCars = new ArrayList();

    // Create the cars in the garage.
    public Garage()
    {
        // Recall, we updated the ctor to set isDirty and shouldRotate.
```

```
            theCars.Add(new Car("Viper", 100, 0, true, false));
            theCars.Add(new Car("Fred", 100, 0, false, false));
            theCars.Add(new Car("BillyBob", 100, 0, false, true));
            theCars.Add(new Car("Bart", 100, 0, true, true));
            theCars.Add(new Car("Stan", 100, 0, false, true));
        }

        // This method takes a Car.CarDelegate as a parameter.
        // Therefore!  'proc' is nothing more than a function pointer!
        public void ProcessCars(Car.CarDelegate proc)
        {
            // Diagnostics: Where are we forwarding the call?
            Console.WriteLine("***** Calling: {0} *****",
                              d.Method.ToString());


            // Diagnostics: Are we calling an instance method or a static
method?
            if(proc.Target != null)
                Console.WriteLine("-->Target: {0}", proc.Target.ToString());
            else
                Console.WriteLine("-->Target is a static method");


            // Real Work: Now call the method, passing in each car.
            foreach(Car c in theCars)
                proc(c);
        }
    }
```

When the object user calls ProcessCars(), it will send in the name of the method that should handle this request. For the sake of argument, assume these are static members named WashCar() and RotateTires(). Consider the following usage:

```
// The garage delegates all work orders to these static functions
// (finding a good mechanic is always a problem…)
public class CarApp
{
    // A target for the delegate.
    public static void WashCar(Car c)
    {
        if(c.Dirty)
            Console.WriteLine("Cleaning a car");
        else
            Console.WriteLine("This car is already clean...");
    }
```

```
        // Another target for the delegate.
        public static void RotateTires(Car c)
        {
            if(c.Rotate)
                Console.WriteLine("Tires have been rotated");
            else
                Console.WriteLine("Don't need to be rotated...");

        }

        public static int Main(string[] args)
        {
            // Make the garage.
            Garage g = new Garage();

            // Wash all dirty cars.
            g.ProcessCars(new Car.CarDelegate(WashCar));

            // Rotate the tires.
            g.ProcessCars(new Car.CarDelegate(RotateTires));

            return 0;
        }
    }
```

Notice (of course) that the two static methods are an exact match to the delegate type (void return value and a single Car argument). Also, recall that when you pass in the name of your function as a constructor parameter, you are adding this item to the internal linked list maintained by System.MulticastDelegate. Figure 5-5 shows the output of this test run. (Notice the output messages supplied by Target and Method properties.)

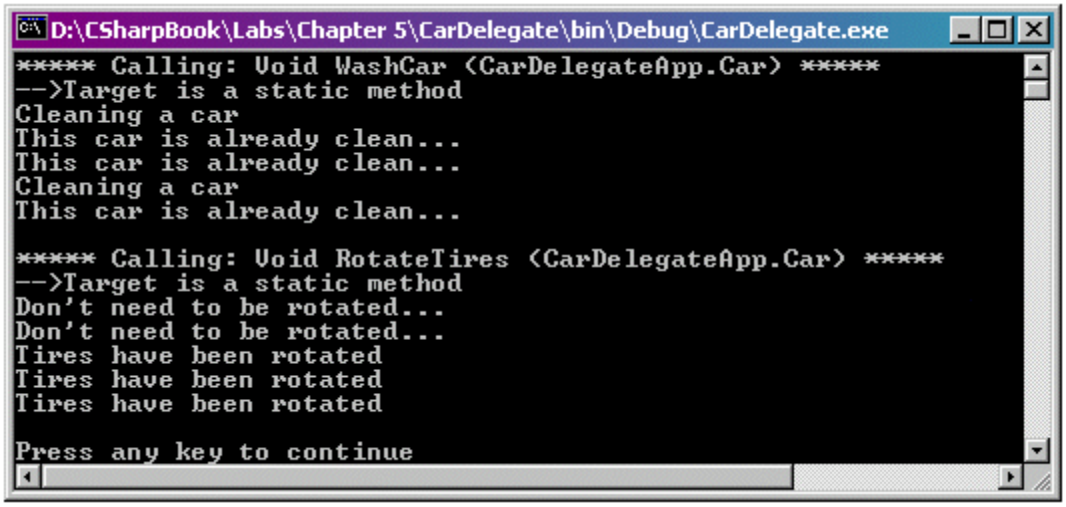

Figure 5-5. Delegate output, take one

Analyzing the Delegation Code

As you can see, the Main() method begins by creating an instance of the Garage type. This class has been configured to delegate all work to other named static functions. Now, when you write the following:

```
// Wash all dirty cars.
g.ProcessCars(new Car.CarDelegate(WashCar));
```

what you are effectively saying is "Add a pointer to the WashCar() function to the CarDelegate type, and pass this delegate to Garage.ProcessCars()." Like most real-world garages, the real work is delegated to another part of the system (which explains why a 30-minute oil change takes 2 hours). Given this, you can assume that ProcessCars() *actually* looks like the following under the hood:

```
// CarDelegate points to the WashCar function:
public void ProcessCars(Car.CarDelegate proc)
{
…
     foreach(Car c in theCars)
          proc(c);          // proc(c) => CarApp.WashCar(c)
…
}
```

Likewise, if you say:

```
// Rotate the tires.
g.ProcessCars(new Car.CarDelegate(RotateTires));
```

ProcessCars() can be understood as:

```
// CarDelegate points to the RotateTires function:
public void ProcessCars(Car.CarDelegate proc)
{
     foreach(Car c in theCars)
          proc(c);          // proc(c) => CarApp.RotateTires(c)
…
}
```

Also notice that when you are calling ProcessCars(), you must create a new instance of the custom delegate:

```
// Wash all dirty cars.
g.ProcessCars(new Car.CarDelegate(WashCar));
// Rotate the tires.
g.ProcessCars(new Car.CarDelegate(RotateTires));
```

This might seem odd at first, given that a delegate represents a function pointer. However, remember that this function pointer is represented by an instance of type System.MulticastDelegate, and therefore must be "new-ed."

Multicasting

Recall that a multicast delegate is an object that is capable of calling any number of functions. In the current example, you did not make use of this feature. Rather, you made two calls to Garage.ProcessCars(), sending in a new instance of the CarDelegate each time. To illustrate multicasting, assume you have updated Main() to look like the following:

```
// Add two function pointers to the internal linked list.
public static int Main(string[] args)
{
    // Make the garage.
    Garage g = new Garage();

    // Create two new delegates.
    Car.CarDelegate wash = new Car.CarDelegate(WashCar);
    Car.CarDelegate rotate = new Car.CarDelegate(RotateTires);

    // The overloaded + operator can be applied to multicast delegates.
    // The result is a new delegate that maintains pointers to
    // both functions.
    g.ProcessCars(wash + rotate);
    return 0;
}
```

Here, you begin by creating two new CarDelegate objects, each of which points to a given function. When you call ProcessCars(), you are actually passing in a new delegate, which holds each function pointer within the internal linked list (crazy huh?). Do note that the + operator is simply a shorthand for calling the static Delegate.Combine() method. Thus, you could write the following equivalent (but uglier) code:

```
// The + operator has the same effect as calling the Combine() method.
g.ProcessCars((Car.CarDelegate)Delegate.Combine(wash, rotate));
```

Furthermore, if you wish to hang on to the new delegate for later use, you could write the following instead:

```
// Create two new delegates.
Car.CarDelegate wash = new Car.CarDelegate(WashCar);
Car.CarDelegate rotate = new Car.CarDelegate(RotateTires);

// Store the new delegate for later use.
MulticastDelegate d = wash + rotate;

// Send the new delegate into the ProcessCars() method.
g.ProcessCars((Car.CarDelegate)d);
```

Regardless of how you configure a multicast delegate, understand that when you call Combine() (or use the overloaded + operator) you are adding a new function pointer to the internal list. If you wish to remove an item from this internal linked list, you can call

the static Remove() method. The first parameter marks the delegate you wish to manipulate, while the second parameter marks the item to remove:

```
// The static Remove() method returns a Delegate type.
Delegate washOnly = MulticastDelegate.Remove(d, rotate);
g.ProcessCars((Car.CarDelegate)washOnly);
```

Before you view the output of this program, let's also update ProcessCars() to print out each function pointer stored in the linked list using Delegate.GetInvocationList(). This method returns an array of Delegate objects, which you iterate over using foreach:

```
// Now print out each member in the linked list.
public void ProcessCarsCar.(CarDelegate proc)
{
    // Where are we passing the call?
    foreach(Delegate d in proc.GetInvocationList())
    {
        Console.WriteLine("***** Calling: " +
                          d.Method.ToString() + " *****");
    }
…
}
```

The output is shown in Figure 5-6.

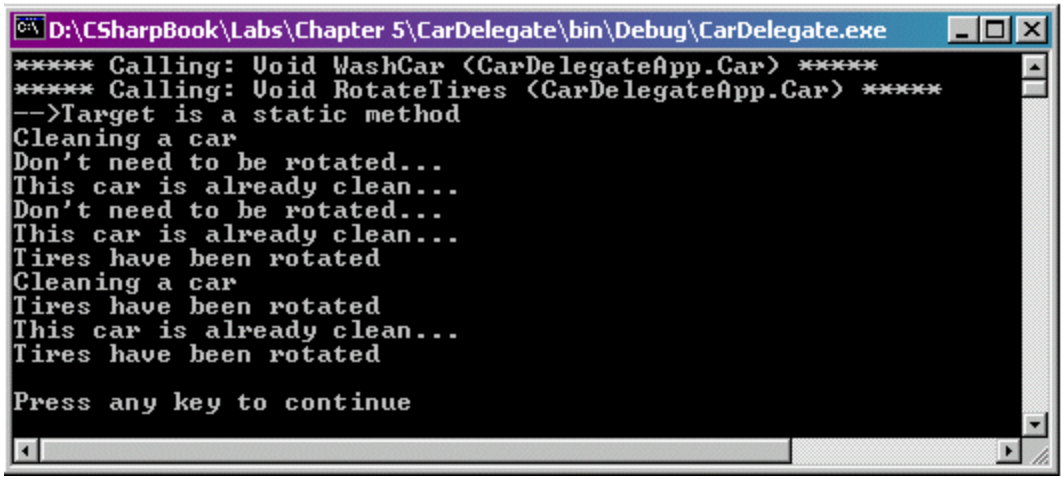

Figure 5-6. Delegate output, take two

Instance Methods as Callbacks

Currently, the CarDelegate type is storing pointers to *static functions*. This is not a requirement of the delegate protocol. It is also possible to delegate a call to a method defined on any *object instance*. To illustrate, assume that the WashCar() and RotateTires() methods have now been moved into a new class named ServiceDept:

```
// We have now moved the static functions into a helper class.
public class ServiceDept
{
    // Not static!
    public void WashCar(Car c)
    {
        if(c.Dirty)
            Console.WriteLine("Cleaning a car");
        else
            Console.WriteLine("This car is already clean...");
    }

    // Still not static!
    public void RotateTires(Car c)
    {
        if(c.Rotate)
            Console.WriteLine("Tires have been rotated");
        else
            Console.WriteLine("Don't need to be rotated...");

    }
}
```

You could now update Main() as so:

```
// Delegate to instance methods of the ServiceDept type.
public static int Main(string[] args)
{
    // Make the garage.
    Garage g = new Garage();

    // Make the service department.
    ServiceDept sd = new ServiceDept();

    // The garage delegates the work to the service department.
    Car.CarDelegate wash = new Car.CarDelegate(sd.WashCar);
    Car.CarDelegate rotate = new Car.CarDelegate(sd.RotateTires);
    MulticastDelegate d = wash + rotate;

    // Tell the garage to do some work.
    g.ProcessCars((Car.CarDelegate)d);

    return 0;
}
```

Now notice the output in Figure 5-7 (check out the name of the target).

```
D:\CSharpBook\Labs\Chapter 5\CarDelegate\bin\Debug\CarDelegate.exe
***** Calling: Void WashCar (CarDelegateApp.Car) *****
***** Calling: Void RotateTires (CarDelegateApp.Car) *****
-->Target: CarDelegateApp.ServiceDept
Cleaning a car
Don't need to be rotated...
This car is already clean...
Don't need to be rotated...
This car is already clean...
Tires have been rotated
Cleaning a car
Tires have been rotated
This car is already clean...
Tires have been rotated

Press any key to continue
```

Figure 5-7. Delegating to instance methods

SOURCE CODE The CarDelegate project is located under the Chapter 5 subdirectory.

Understanding (and Using) Events

Delegates are fairly interesting constructs because you can resolve the name of a function to call at runtime, rather than compile time. Admittedly, this syntactic orchestration can take a bit of getting used to. However, because the ability for one object to call back to another object is such a helpful construct, C# provides the "event" keyword to lessen the burden of using delegates in the raw.

The most prevalent use of the event keyword would be found in GUI-based applications, in which Button, TextBox, and Calendar widgets all report back to the containing Form when a given action (such as clicking a Button) has occurred. However, events are not limited to GUI-based applications. Indeed, they can be quite helpful when creating "non-GUI" based projects (as you will now see).

Recall that the current implementation of Car.SpeedUp() (see Chapter 3) throws an exception if the user attempts to increase the speed of an automobile that has already been destroyed. This is a rather brute force way to deal with the problem, given that the exception has the potential to halt the program's execution if the error is not handled in an elegant manner. A better design would be to simply inform the object user when the car has died using a custom event, and allow the caller to act accordingly.

Let's reconfigure the Car to send two events to those who happen to be listening. The first event (AboutToBlow) will be sent when the current speed is 10 miles below the maximum speed. The second event (Exploded) will be sent when the user attempts to speed up a car that is already dead. Establishing an event is a two-step process. First, you need to define a delegate, which as you recall represents a pointer to the method(s) to call when the event is sent. Next, you define the events themselves using the "event" keyword. Here is the updated Car class (also notice that I have added a private Boolean to represent the state of the car):

```
// This car can 'talk back' to the user.
public class Car
{
…
     // Is the car alive or dead?
     private bool dead;

     // Holds the function(s) to call when the event occurs.
     public delegate void EngineHandler(string msg);

     // This car can send these events.
     public static event EngineHandler Exploded;
     public static event EngineHandler AboutToBlow;
 …
}
```

Firing an event (i.e., sending the event to those who happen to be listening) is as simple as specifying the event by name and sending out any specified parameters. To illustrate, update the previous implementation of SpeedUp() to send each event accordingly (and remove the previous exception logic):

```
// Fire the correct event based on our current state of affairs.
public void SpeedUp(int delta)
{
     // If the car is dead, send exploded event.
     if(dead)
     {
          if(Exploded != null)
               Exploded("Sorry, this car is dead...");
     }
     else
     {
          currSpeed += delta;

          // Almost dead?  Send about to blow event.
          if(10 = = maxSpeed - currSpeed)
               if(AboutToBlow != null)
                    AboutToBlow("Careful, approaching terminal speed!");

          // Still OK!  Proceed as usual.
          if(currSpeed >= maxSpeed)
               dead = true;
          else
               Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
     }
}
```

With this, you have configured the car to send two custom events (under the correct conditions). You will see the usage of this new automobile in just a moment, but first, let's check the event architecture in a bit more detail.

Events Under the Hood

A given event actually expands into two hidden public functions, one having an "add_" prefix, the other having a "remove_" prefix. For example, the Exploded event expands to the following methods:

```
// The following event expands to:
// add_Exploded()
// remove_Exploded()
//
public static event EngineHandler Exploded;
```

In addition to defining hidden add_XXX() and remove_XXX() methods, each event also actually maps to a private static class, which associates the corresponding delegate to a given event. In this way, when an event is raised, each method maintained by the delegate will be called. This is a convenient way to allow an object to broadcast the event to multiple "event sinks."

To illustrate, check out Figure 5-8, a screenshot of the Car type as seen through the eyes of ILDasm.exe.

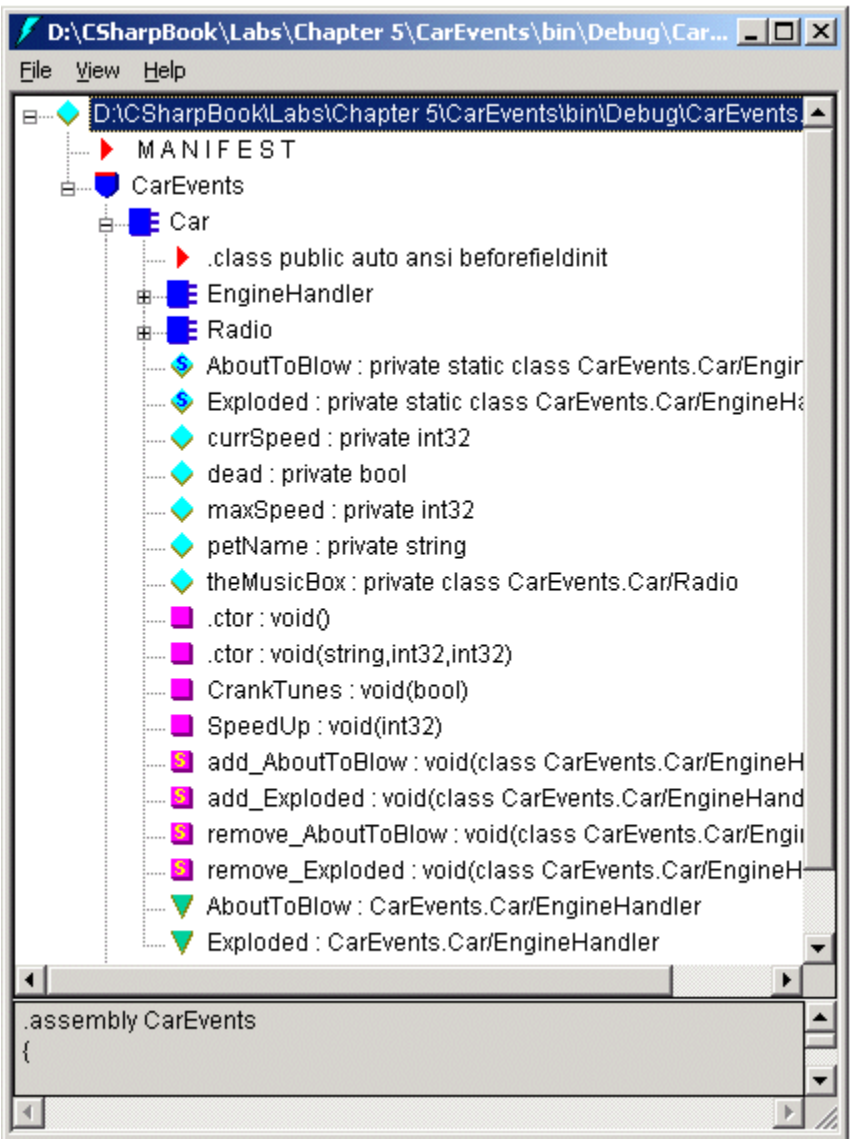

Figure 5-8. Events under the hood

As you can see, each event (Exploded and AboutToBlow) is internally represented as the following members:

* A private static class
* An add_XXX() method
* A remove_XXX() method

If you were to check out the IL instructions behind add_AboutToBlow(), you would find the following (note the call to Delegate.Combine() is handled on your behalf):

```
.method public hidebysig specialname static
void  add_AboutToBlow(class CarEvents.Car/EngineHandler 'value') cil managed
synchronized
{
```

```
  // Code size       22 (0x16)
  .maxstack  8
  IL_0000:  ldsfld     class CarEvents.Car/EngineHandler
CarEvents.Car::AboutToBlow
  IL_0005:  ldarg.0
  IL_0006:  call       class [mscorlib]System.Delegate
  [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
  class [mscorlib]System.Delegate)
  IL_000b:  castclass  CarEvents.Car/EngineHandler
  IL_0010:  stsfld     class CarEvents.Car/EngineHandler
CarEvents.Car::AboutToBlow
  IL_0015:  ret
} // end of method Car::add_AboutToBlow
```

As you would expect, remove_AboutToBlow() will make the call to Delegate.Remove() automatically:

```
.method public hidebysig specialname static
     void  remove_AboutToBlow(class CarEvents.Car/EngineHandler 'value')
                                                    cil managed synchronized
{
  // Code size         22 (0x16)
  .maxstack  8
  IL_0000:  ldsfld     class CarEvents.Car/EngineHandler
CarEvents.Car::AboutToBlow
  IL_0005:  ldarg.0
  IL_0006:  call       class [mscorlib]System.Delegate
[mscorlib]System.Delegate::Remove(class [mscorlib]System.Delegate,
                                  class [mscorlib]System.Delegate)
  IL_000b:  castclass  CarEvents.Car/EngineHandler
  IL_0010:  stsfld     class CarEvents.Car/EngineHandler
CarEvents.Car::AboutToBlow
  IL_0015:  ret
} // end of method Car::remove_AboutToBlow
```

The IL instructions for the event itself make use of the [.addon] and [.removeon] tags to establish the correct add_XXX and remove_XXX methods (also note the static private class is mentioned by name):

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
  .addon void CarEvents.Car::add_AboutToBlow(class CarEvents.Car/EngineHandler)
  .removeon
         void CarEvents.Car::remove_AboutToBlow(class
CarEvents.Car/EngineHandler)
} // end of event Car::AboutToBlow
```

So, now that you understand how to build a class that can send events, the next big question is how you can configure an object to receive these events.

Listening to Incoming Events

Assume you have now created an instance of the Car class and wish to listen to the events it is capable of sending. The goal is to create a method that represents the "event sink" (i.e., the method called by the delegate). To do so, you need to call the correct add_XXX() method to ensure that your method is added to the list of function pointers maintained by your delegate. However, you do not call add_XXX() and remove_XXX() directly, but rather use the overloaded += and -= operators. Basically, when you wish to listen to an event, follow the pattern shown here:

```
// I'm listening…
// ObjectVariable.EventName += new ObjectVariable.DelegateName(functionToCall);
//
Car.Exploded += new Car.EngineHandler(OnBlowUp);
```

When you wish to detach from a source of events, use the -= operator:

```
// Shut up already!
// ObjectVariable.EventName -= new ObjectVariable.DelegateName(functionToCall);
//
Car.Exploded -= new Car.EngineHandler(OnBlowUp);
```

Here is a complete example (output is shown in Figure 5-9):

```
// Make a car and listen to the events.
public class CarApp
{
     public static int Main(string[] args)
     {
          Car c1 = new Car("SlugBug", 100, 10);

          // Hook into events.
          Car.Exploded += new Car.EngineHandler(OnBlowUp);
          Car.AboutToBlow += new Car.EngineHandler(OnAboutToBlow);

          // Speed up (this will generate the events.)
          for(int i = 0; i < 10; i++) c1.SpeedUp(20);

          // Detach from events.
          Car.Exploded -= new Car.EngineHandler(OnBlowUp);
          Car.Exploded -= new Car.EngineHandler(OnAboutToBlow);

          // No response!
          for(int i = 0; i < 10; i++) c1.SpeedUp(20);
          return 0;
     }

     // OnBlowUp event sink.
     public static void OnBlowUp(string s)
     {
```

```
            Console.WriteLine("Message from car: {0}", s);
        }
        // OnAboutToBlow event sink.
        public static void OnAboutToBlow(string s)
        {
            Console.WriteLine("Message from car: {0}", s);
        }
    }
```

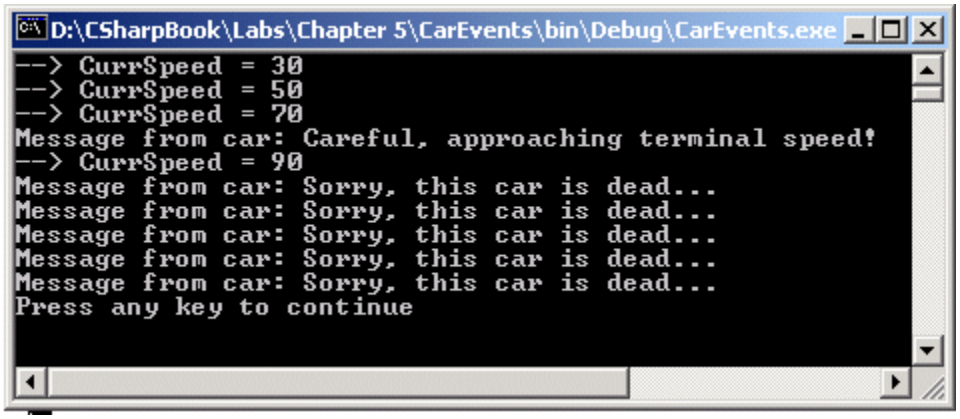

Figure 5-9. Handling your Car's event set

If you wish to have multiple event sinks called by a given event, simply repeat the process:

```
// Multiple event sinks.
public class CarApp
{
    public static int Main(string[] args)
    {
        // Make a car as usual.
        Car c1 = new Car("SlugBug", 100, 10);

        // Hook into events.
        Car.Exploded += new Car.EngineHandler(OnBlowUp);
        Car.Exploded += new Car.EngineHandler(OnBlowUp2);
        Car.AboutToBlow += new Car.EngineHandler(OnAboutToBlow);

        // Speed up (this will generate the events.)
        for(int i = 0; i < 10; i++)
            c1.SpeedUp(20);

        // Detach from events.
        Car.Exploded -= new Car.EngineHandler(OnBlowUp);
        Car.Exploded -= new Car.EngineHandler(OnBlowUp2);
        Car.Exploded -= new Car.EngineHandler(OnAboutToBlow);
```

```
        …
    }


    // OnBlowUp event sink A.
    public static void OnBlowUp(string s)
    {
        Console.WriteLine("Message from car: {0}", s);
    }

    // OnBlowUp event sink B.
    public static void OnBlowUp2(string s)
    {
        Console.WriteLine("-->AGAIN I say: {0}", s);
    }

    // OnAboutToBlow event sink.
    public static void OnAboutToBlow(string s)
    {
            Console.WriteLine("Message from car: {0}", s);
    }
}
```

Now, when the Exploded event is sent, the associated delegate calls OnBlowUp() as well as OnBlowUp2(), as shown in Figure 5-10.

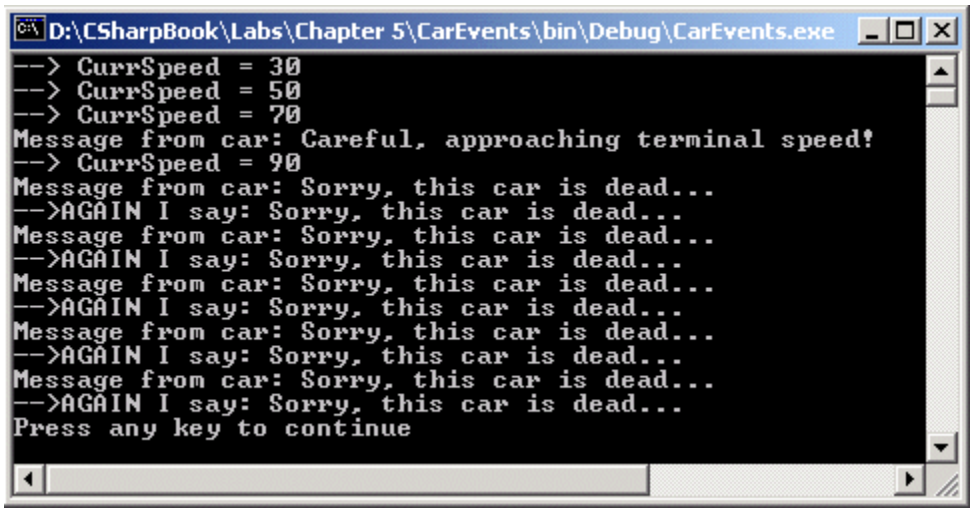

Figure 5-10. Working with multiple event handlers

Objects as Event Sinks

At this point, you have the background to build objects that can participate in a two-way conversation. However, understand that you are free to build a helper object to respond to

an object's event set, much in the same way that you created a helper class to be called by all delegates. For example, let's move your event sink methods out of the CarApp class and into a new class named CarEventSink:

```
// Car event sink
public class CarEventSink
{
     // OnBlowUp event handler.
     public void OnBlowUp(string s)
     {
          Console.WriteLine("Message from car: {0}", s);
     }

     // OnBlowUp event handler version 2.
     public void OnBlowUp2(string s)
     {
          Console.WriteLine("-->AGAIN I say: {0}", s);
     }

     // OnAboutToBlow handler.
     public void OnAboutToBlow(string s)
     {
          Console.WriteLine("Message from car: {0}", s);
     }
}
```

The CarApp class is then a bit more self-contained, as the event sink methods have been pulled out of the CarApp definition and into their own custom type. Here is the update:

```
// Note the creation and use of the CarEventSink.
public class CarApp
{
     public static int Main(string[] args)
     {
          Car c1 = new Car("SlugBug", 100, 10);

          // Make the sink object.
          CarEventSink sink = new CarEventSink();

          // Hook into events using sink object.
          Car.Exploded += new Car.EngineHandler(sink.OnBlowUp);
          Car.Exploded += new Car.EngineHandler(sink.OnBlowUp2);
          Car.AboutToBlow += new Car.EngineHandler(sink.OnAboutToBlow);

          for(int i = 0; i < 10; i++)
               c1.SpeedUp(20);
```

```
        // Detach from events using sink object.
        Car.Exploded -= new Car.EngineHandler(sink.OnBlowUp);
        Car.Exploded -= new Car.EngineHandler(sink.OnBlowUp2);
        Car.Exploded -= new Car.EngineHandler(sink.OnAboutToBlow);

        return 0;
    }
}
```

The output is (of course) identical.

SOURCE CODE The CarEvents project is located under the Chapter 5 subdirectory.

Designing an Event Interface

COM programmers may be familiar with the notion of defining and implementing "callback interfaces." This technique allows a COM client to receive events from a coclass using a custom COM interface, and is often used to bypass the overhead imposed by the official COM connection point architecture. For an illustration of using the interface as a callback, let's examine how callback interfaces can be created using C# (and .NET in general). Consider this last topic a bonus section, which proves the point that there is always more than one way to solve a problem.

First, let's keep the same assumption that the Car type wishes to inform the outside world when it is about to blow (current speed is 10 miles below the maximum speed) and has exploded. However, this time you will *not* be using the "delegate" or "event" keywords, but rather the following custom interface:

```
// The engine event interface.
public interface IEngineEvents
{
    void AboutToBlow(string msg);
    void Exploded(string msg);
}
```

This interface will be implemented by a sink object, on which the Car will make calls. Here is a sample implementation:

```
// Car event sink.
public class CarEventSink : IEngineEvents
{
    public void AboutToBlow(string msg)
    {
        Console.WriteLine(msg);
    }

    public void Exploded(string msg)
    {
        Console.WriteLine(msg);
    }
```

```
}
```

Now that you have an object that implements the event interface, your next task is to pass a reference to this sink into the Car. The Car holds onto the reference, and makes calls back on the sink when appropriate. In order to allow the Car to obtain a reference to the sink, you can assume some method has been added to the default public interface.

In keeping with the COM paradigm, let's call this method Advise(). When the object user wishes to detach from the event source, he may call another method (Unadvise() in COM-speak). In order to allow the object user to register multiple event sinks, let's assume that the Car maintains an ArrayList to represent each outstanding connection (analogous to the array of IUnknown* interfaces used with classic COM connection points). Here is the story so far:

```
// This Car does not make any use of C# delegates or events.
public class Car
{
     // The set of connected sinks.
     ArrayList itfConnections = new ArrayList();

     // Attach or disconnect from the source of events.
     public void Advise(IEngineEvents itfClientImpl)
     {
          itfConnections.Add(itfClientImpl);
     }

     public void Unadvise(IEngineEvents itfClientImpl)
     {
          itfConnections.Remove(itfClientImpl);
     }
…
}
```

Now, Car.SpeedUp() can be retrofitted to iterate over the list of connections and fire the correct notification when appropriate (i.e., call the correct method on the sink):

```
// Interface based event protocol!
//
class Car
{
…
     public void SpeedUp(int delta)
     {
          // If the car is dead, send exploded event to each sink.
          if(dead)
          {
               foreach(IEngineEvents e in itfConnections)
                    e.Exploded("Sorry, this car is dead...");
          }
          else
```

```
        {
            currSpeed += delta;

            // Dude, you're almost dead!  Proceed with caution!
            if(10 = = maxSpeed - currSpeed)
            {
                foreach(IEngineEvents e in itfConnections)
                    e.AboutToBlow("Careful buddy!  Gonna blow!");
            }

            // Still OK!
            if(currSpeed >= maxSpeed)
                dead = true;
            else
                Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
        }
}
```

The following is some client-side code, now making use of a callback interface to listen to the Car events:

```
// Make a car and listen to the events.
public class CarApp
{
    public static int Main(string[] args)
    {
        Car c1 = new Car("SlugBug", 100, 10);

        // Make sink object.
        CarEventSink sink = new CarEventSink();

        // Pass the Car a reference to the sink.
        // (The lab solution registers multiple sinks…).
        c1.Advise(sink);

        // Speed up (this will generate the events.)
        for(int i = 0; i < 10; i++)
            c1.SpeedUp(20);

        // Detach from events.
        c1.Unadvise(sink);
        return 0;
    }
}
```

The output should look very familiar (see Figure 5-11).

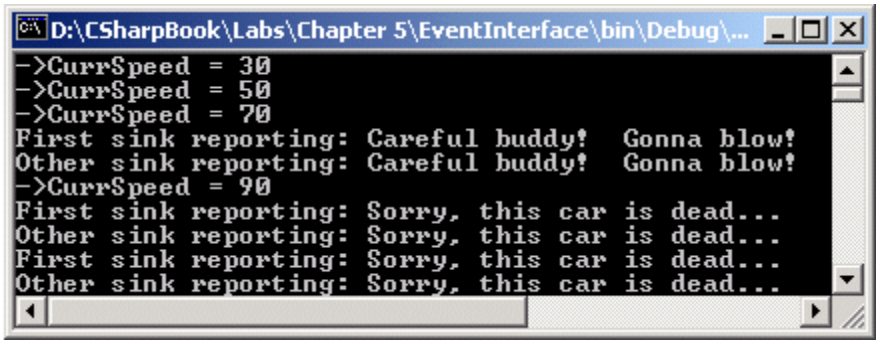

Figure 5-11. Interfaces as an event protocol

SOURCE CODE The EventInterface project is located under the Chapter 5 subdirectory.

XML-Based Documentation

This final topic of this chapter is by no means as mentally challenging as the .NET delegation protocol, and is not necessarily an "advanced" technique. Nevertheless, your next goal is to examine a technique provided by C#, which enables you to turn your source code documentation into a corresponding XML file. If you have a background in Java, you are most likely familiar with the javadoc utility. Using javadoc, you are able to turn Java source code into an HTML representation. The C# documentation model is slightly different, in that the "source code to XML formatting" process is the job of the C# compiler (csc.exe) rather than a standalone utility.

So, why use XML to represent your type definitions rather than HTML? The primary reason is that XML is a very enabling technology. Given that XML separates raw data from the presentation of that data, you (as a programmer) can apply any number of XML transformations to the raw XML. As well, you could programmatically read the XML file using types defined in the .NET base class library.

When you wish to document your types in XML, your first step is to make use of a special comment syntax, the triple forward slash (///) rather than the C++ style double slash (//) or C-based (/*… */) syntax. After the triple slash, you are free to use any well-formed XML tags, including the following predefined set (see Table 5-3).

Table 5-3. Stock XML Tags

Predefined XML Documentation Tag	**Meaning in Life**
<c>	Indicates that text within a description should be marked as code
<code>	Indicates multiple lines should be marked as code
<example>	Used to mock up a code example for the item you are describing
<exception>	Used to document which exceptions a given class may throw
<list>	Used to insert a list into the documentation file
<param>	Describes a given parameter
<paramref>	Associates a given XML tag with a specific parameter
<permission>	Used to document access permissions for a member
<remarks>	Used to build a description for a given member
<returns>	Documents the return value of the member
<see>	Used to cross-reference related items
<seealso>	Used to build an "also see" section within a description
<summary>	Documents the "executive summary" for a given item
<value>	Documents a given property

The following is a very streamlined Car type with some XML-based comments. In particular, note the use of the <summary> and <param> tags:

```
/// <summary>
///    This is a simple Car that illustrates
///    working with XML style documentation.
/// </summary>
public class Car
{
    /// <summary>
    /// Do you have a sunroof?
    /// </summary>
    private bool hasSunroof = false;

    /// <summary>
    /// The ctor lets you set the sunroofedness.
    /// </summary>
    /// <param name="hasSunroof"> </param>
    public Car(bool hasSunroof)
    {
        this.hasSunroof = hasSunroof;
```

```
        }

        /// <summary>
        /// This method allows you to open your sunroof.
        /// </summary>
        /// <param name="state"> </param>
        public void OpenSunroof(bool state)
        {
            if(state = = true && hasSunroof = = true)
            {
                Console.WriteLine("Put sunscreen on that bald head!");
            }
            else
            {
                Console.WriteLine("Sorry...you don't have a sunroof.");
            }
        }
        /// <summary>
        /// Entry point to application.
        /// </summary>
        public static void Main()
        {
            SimpleCar c = new SimpleCar(true);
            c.OpenSunroof(true);
        }
    }
```

Once you have your XML documentation in place, you can specify the /doc flag as input to the C# compiler. Note that you must specify the name of the XML output file as well as the C# input file:

```
csc /doc:simplecar.xml simplecar.cs
```

As you would hope, the Visual Studio.NET IDE enables you to specify the name of an XML file to describe your types. To do so, click the Properties button from the Solution Explorer window (see Figure 5-12).

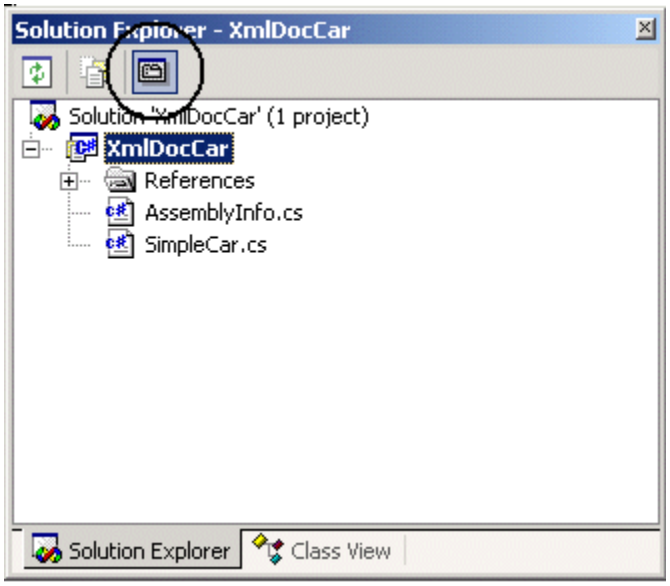

Figure 5-12. Activating the Project Properties dialog

Once you've activated the Project Properties dialog, select the Build option from the Configuration Properties folder. Here you will find an edit box (XML Documentation File) that enables you to specify the name of the file that will contain XML definitions for the types in your project (which is automatically regenerated as you rebuild your project).

Viewing the Generated XML File

If you were now to open the simplecar.xml file from within the Visual Studio.NET IDE, you would find the display shown in Figure 5-13.

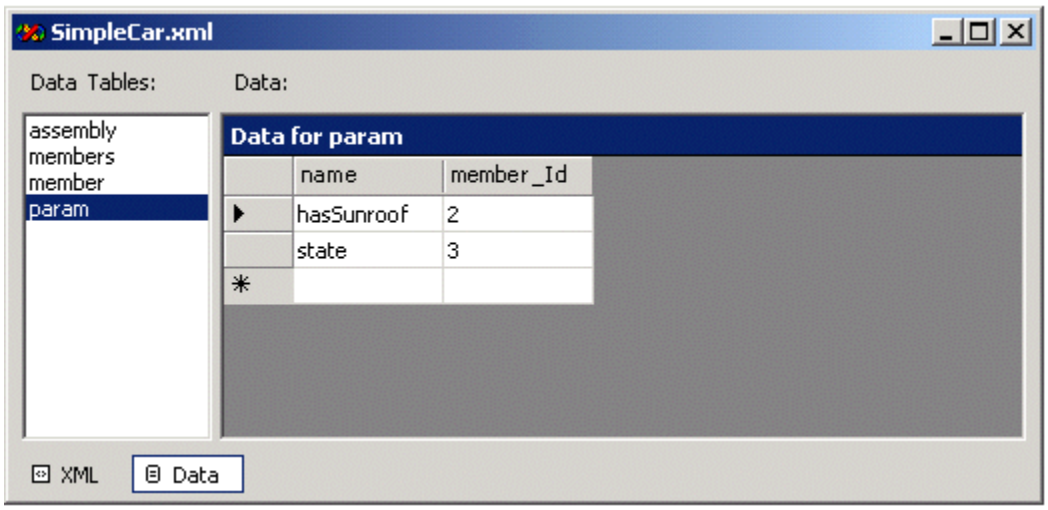

Figure 5-13. The Visual Studio.NET XML viewer

If you were to select the XML button from the XML editor window, you would find the raw XML format. Be aware that assembly members are denoted with the <member>

tag, fields are marked with an F prefix, types with T, and members with M. Table 5-4 provides some additional XML format characters.

Table 5-4. XML Format Characters

Format Character	**Meaning in Life**
N	Denotes a namespace
T	Represents a type (i.e., class, interface, struct, enum, delegate)
F	Represents a field
P	Represents type properties (including indexers)
M	Represents method (including such constructors and overloaded operators)
E	Denotes an event
!	Represents an error string that provides information about the error. The C# compiler generates error information for links that cannot be resolved.

At this point, you have a raw XML file that can be rendered into HTML using an XSL style sheet or programmatically manipulated using .NET types. Although this approach gives you the biggest bang for the buck when it comes to customizing the look and feel of your source code comments, there is another alternative.

Visual Studio.NET Documentation Support

If the thought of ending up with a raw XML file is a bit anticlimactic, be aware that VS.NET does offer another comment-formatting option. Using the same XML tags you have just examined, you may make use of the "Tools | Build Comment Web Pages…" menu option. When you select this item, you will be asked if you wish to build the entire solution or a specific project within the solution set, as shown in Figure 5-14.

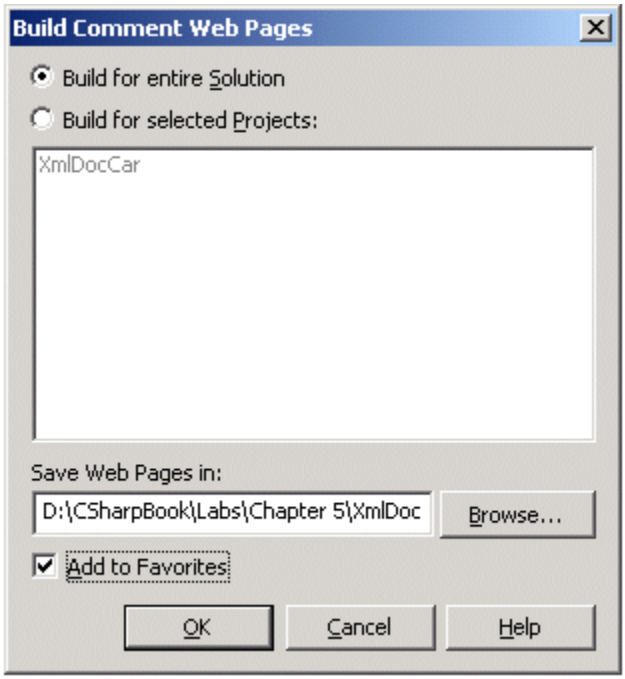

Figure 5-14. Configuration of your HTML-based documentation

The Build Comment Web Pages option will respond by creating a new folder in your project directory that holds a number of images and HTML files built based on your XML documentation. You can now open the main HTML file and view your commented project. For example, check out Figure 5-15:

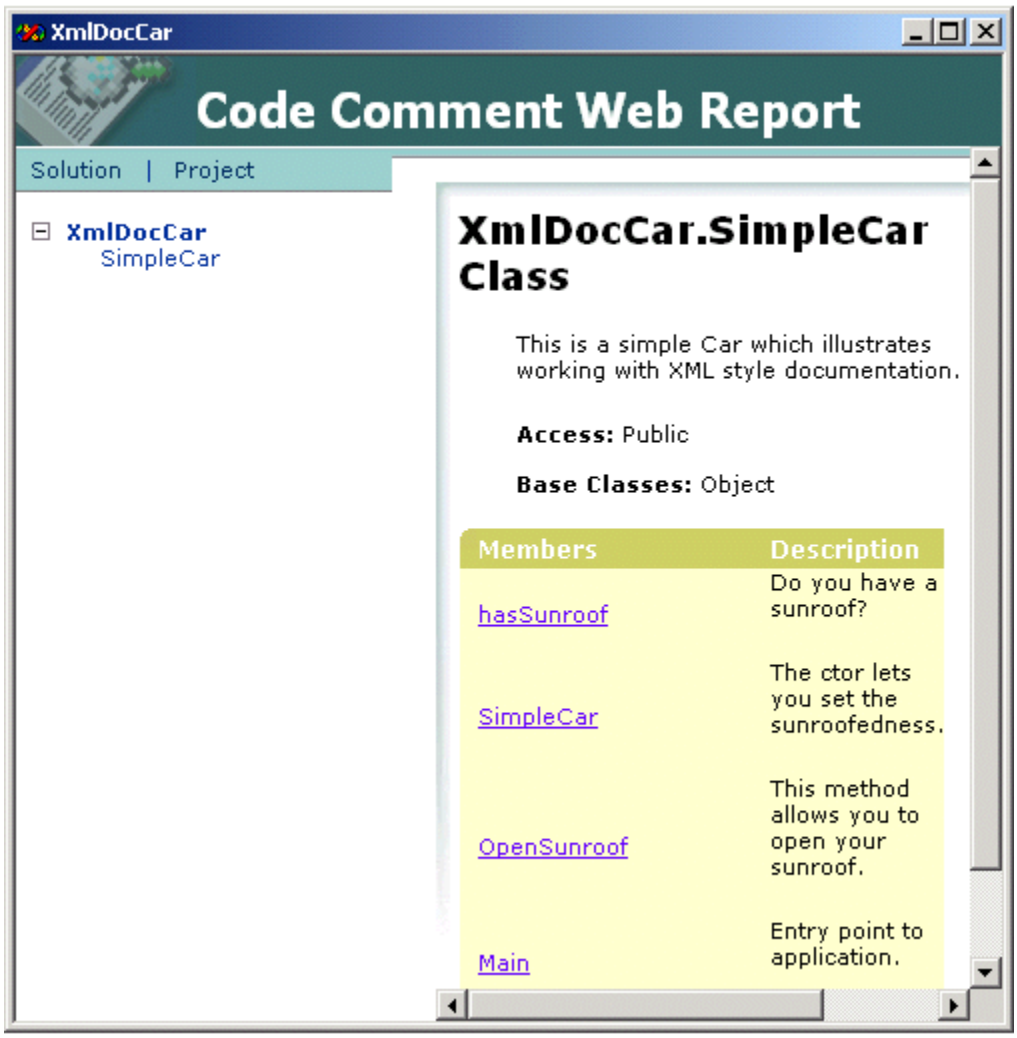

Figure 5-15. The generated XmlCarDoc online documentation

SOURCE CODE The XmlDocCar project is located under the Chapter 5 subdirectory.

Summary

The purpose of this chapter was to round out your understanding of the key features of the C# language. You are now well-equipped to build sophisticated object models that function well within the .NET universe. The chapter began by examining how to build a custom indexer method, which allows the object user to access discrete sub-items using array-like notation. Next, the chapter examined how the C# language enables you to overload various operators in order to let your custom types behave a bit more intuitively to the object users of the world.

You have also seen three ways in which multiple objects can partake in a bidirectional conversation. The first two approaches (delegates and events) are official, well-supported constructs in the .NET universe. The third approach (event interfaces) is more of a design pattern than a language protocol; however, it does allow two entities to communicate in a type-safe manner.

I wrapped up this chapter by examining how to comment your types using XML comment tags, and you learned how the Visual Studio.NET IDE can make use of these tags to generate online documentation for your current project. Using these techniques, you enable your peers to fully understand the fruit of your .NET labors.